Intel® Array Building Blocks

Dr.-Ing. Michael Klemm
Sen. Application Engineer
Software and Services Group

Legal Disclaimer

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Optimization Notice

Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the "Intel® Compiler User and Reference Guides" under "Compiler Options." Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101

Agenda

- Overview and Goals
- How to add it to your project…
- Programming Constructs and Data Types

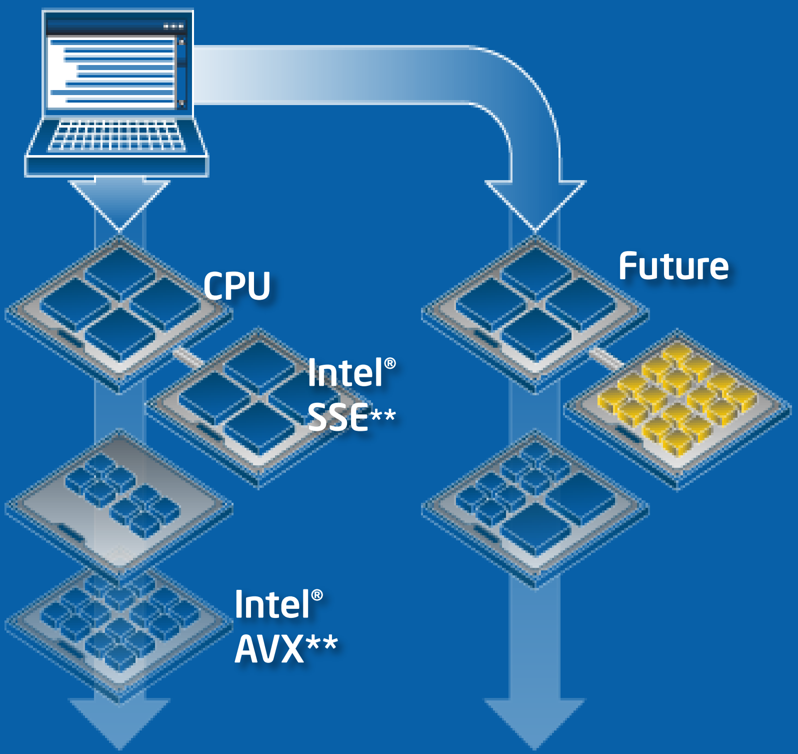
Introduction to Intel® Array Building Blocks

Overview and Goals

Introduction: Objectives

- Understand the motivation for Intel® Array Building Blocks
  - Also known as Intel® ArBB
- Understand the Intel® ArBB C++ API-as-a-language
- Understand the basic syntax of the Intel® ArBB API
- Review the available operators
- Be able to write a first "Hello World" application w/ ArBB
- Work through a few example applications

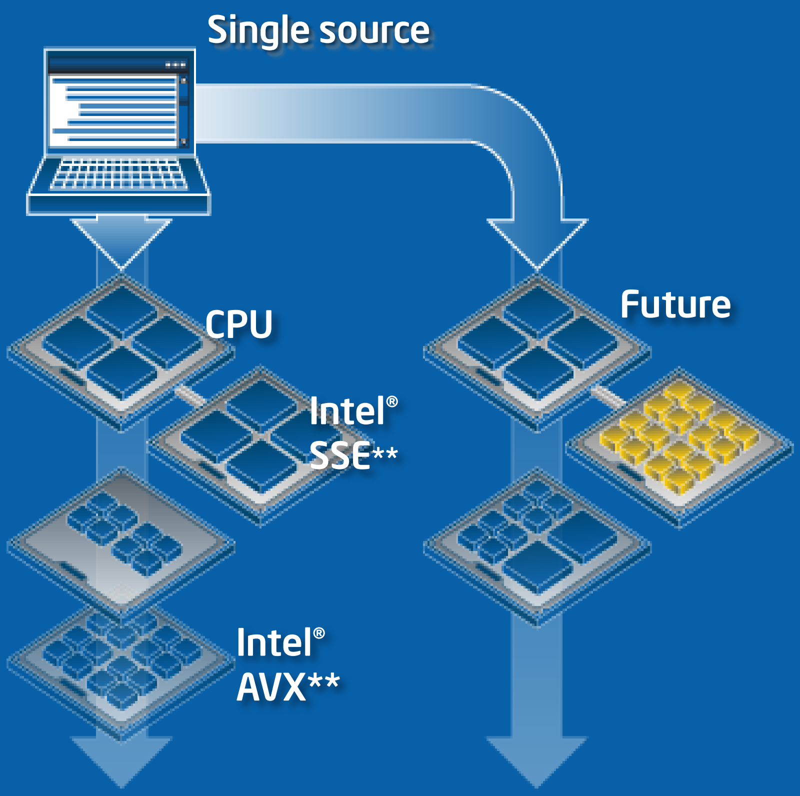
What's Wrong with Parallel Programming?



- **Parallel programming is hard**
  - Deadlocks
  - Data races
  - Synchronization
  - Load imbalance
- **Errors inhibit productivity**
- **No uniform programming model for**
  - Intel SSE, Intel AVX
  - Multi-threading
  - IA manycore
- **Parallel programmers lose single code base for their applications**

** Intel® Streaming SIMD Extensions
Intel® Advanced Vector Extensions

Intel® Array Building Blocks



Productivity
- Integrates with existing tools
- Applicable to many problem domains
- Safe by default → maintainable

Performance
- Efficient and scalable
- Harnesses both vectors and threads
- Eliminates modularity overhead of C++

Portability
- High-level abstraction
- Hardware independent
- Forward scaling

** Intel® Streaming SIMD Extensions
Intel® Advanced Vector Extensions

Productivity

- Integrates
  with existing IDEs, tools, and compilers: no new compiler needed
- Interoperates
  with other Intel parallel programming tools and libraries
- Incremental
  allows selective and targeted modification of existing code bases
- Expressive
  syntax oriented to application experts
- Safe by default
  deterministic semantics avoid race conditions and deadlock by construction
- Easy to learn
  serially consistent semantics and simple interface leverage existing skills
- Widely applicable
  Generalized data parallel model applicable to many types of computations

Performance

- Scalable to large problems
  manages data to directly address memory bottlenecks
- Unified thread and vector parallelization
  single specification targets multiple mechanisms
- Elimination of modularity overhead
  automatically fuses multiple operations
- Wide *and* deep
  developers can choose level of abstraction
  can drill down to detail if needed

Portability

- **High-level**
  avoids dependencies on particular hardware mechanisms or architectures
- **ISA extension independent**
  common binary can exploit different ISA extensions transparently
- **Allows choice of deployment hardware today**
  including scaling to many cores
- **Allows migration and forward-scaling**
  will support future hardware roadmap

ISA: Instruction Set Architecture

Productivity via a High Level of Abstraction

"Specify what to do, not how to do it!"

Mathematical structure
Data organization
Where's my data race?
What caused that deadlock?
Why do I get different answers every time I run this?
How many threads should I use?
How big is my cache?
How do I deal with different ISAs and vector widths?
Where's the guy who originally wrote this thing – I can't figure out what the code is supposed to be computing!

Mathematical structure
Data organization

Goal: increasing the efficiency of the expert application developer

Get the Best of Both Worlds

Attribute	**Productivity** (math/scripting languages)	**Performance** (threads with intrinsics)	**Productivity & Performance** (Array Building Blocks)
Readability Clear and understandable notation	Green	Red	Green
Determinism Output is always the same for a given input	Green	Red	Green
Correctness Major sources of error are avoided	Green	Red	Green
Performance Best absolute performance	Red	Green	Yellow
Scalability Ability to take advantage of increased number of cores	Red	Yellow	Green
Maintainability Easy to maintain	Green	Red	Green

Green: Provides excellent support

Yellow: Provides something in-between

Red: Provides little or no support

Intel® ArBB vs. Intel® SSE intrinsics

ArBB

```
template <typename T>
dense<T> CND(dense<T> x)
{
  dense<T> l = abs(x);
  dense<T> k = 1.0f / ( 1.0f + 0.2316419f * l);

  dense<T> w =
    0.31938153f * k -
    0.356563782f * k * k +
    1.781477937f * k * k * k -
    1.821255978f * k * k * k * k +
    1.330274429f * k * k * k * k * k;

  w = w * inv_sqrt_2xPI * exp(l * l * -0.5f);
  w = select(x > 0, 1.0f - w, w);
  return w;
}

templat[illegible]
void ct[illegible]schol[illegible]n(dense<T> [illegible]
[illegible]se<T> x,
[illegible]se<T> r,
[illegible]
[illegible]dense<T>& res[illegible]
[illegible]
[illegible]se<T> sqrt_value[illegible]rt(t);
    dense<T> d1 = ln(s / x) + (r + v * v * 0.5f ) * t) / sqrt_value;
    dense<T> d2 = d1 - sqrt_value;

    result = x * exp(0f - r * t) * (1.0f - CND(d2)) + (-s) * (1.0 - CND(d1));
}


template <typename T>
void ctBlackScholes(T *option_price,   int num_options,
          T *stkprice,   T *strike,     T *rate,   T *volatility,
          T *time)
{
    dense<T> s(stkprice, num_options);
    dense<T> x(strike, num_options);
    dense<T> r(rate, num_options);
    dense<T> v(volatility, num_options);
    dense<T> t(time, num_options);
    dense<T> result(option_price, num_options);
    call( ctBlackScholesKern<T> )( s,x,r,v,t,result );
}
```

42 lines

- vectorized
- threaded
- machine independent

SSE

```
#define NCO 4

#if (NCO==2)
#define fptype double
#define SIMD_WIDTH 2
#define _MMR     __m128d
#define _MM_LOAD _mm_load_pd
#define _MM_STORE _mm_store_pd
#define _MM_MUL  _mm_mul_pd
#define _MM_ADD  _mm_add_pd
#define _MM_SUB  _mm_sub_pd
#define _MM_DIV  _mm_div_pd
#define _MM_SQRT _mm_sqrt_pd
#define _MM_SET(A) _mm_set_pd(A,A)
#define _MM_SETR _mm_set_pd
#endif

#if (NCO==4)
#define fptype float
#define SIMD_WIDTH 4
#define _MMR     __m128
#define _MM_LOAD _mm_load_ps
#define _MM_STORE _mm_store_ps
#define _MM_MUL  _mm_mul_ps
#define _MM_ADD  _mm_add_ps
#define _MM_SUB  _mm_sub_ps
#define _MM_DIV  _mm_div_ps
#define _MM_SQRT _mm_sqrt_ps
#define _MM_SET(A) _mm_set_ps(A,A,A,A)
#define _MM_SETR _mm_set_ps
#endif

__forceinline void CNDF ( fptype * OutputX, fptype * InputX )
{
  _MM_ALIGN16 int sign[SIMD_WIDTH];
  int i;
  _MMR xInput;
  _MMR xNPrimeofX;
  _MM_ALIGN16 fptype expValues[SIMD_WIDTH];
  _MMR xK2;
  _MMR xK2_2, xK2_3, xK2_4, xK2_5;
  _MMR xLocal, xLocal_1, xLocal_2, xLocal_3;

  for (i=0; i<SIMD_WIDTH; i++) {
    // Check for negative value of Inp[illegible]
    if (InputX[i] < 0.0) {
      InputX[i] = -InputX[i];
      sign[i] = 1;
    } else
      sign[i] = 0;
  }

  xInput = _MM_LOAD(InputX);

  // Compute NPrimeX term common to both four & six decimal
accuracy calcs
  for (i=0; i<SIMD_WIDTH; i++) {
    expValues[i] = exp(-0.5f * InputX[i] * InputX[i]);
    // printf("exp[%d]: %f\n", i, expValues[i]);
  }

  xNPrimeofX = _MM_LOAD(expValues);
  xNPrimeofX = _MM_MUL(xNPrimeofX, _MM_SET(inv_sqrt_2xPI));

  xK2 = _MM_MUL(_MM_SET((fptype)0.2316419), xInput);
  xK2 = _MM_ADD(xK2, _MM_SET((fptype)1.0));
  xK2 = _MM_DIV(_MM_SET((fptype)1.0), xK2);
  // xK2 = _mm_rcp_pd(xK2);  // No rcp function for double-precision

  xK2_2 = _MM_MUL(xK2, xK2);
  xK2_3 = _MM_MUL(xK2_2, xK2);
  xK2_4 = _MM_MUL(xK2_3, xK2);
  xK2_5 = _MM_MUL(xK2_4, xK2);

  xLocal_1 = _MM_MUL(xK2, _MM_SET((fptype)0.319381530));
  xLocal_2 = _MM_MUL(xK2_2, _MM_SET((fptype)-0.356563782));
  xLocal_3 = _MM_MUL(xK2_3, _MM_SET((fptype)1.781477937));
  xLocal_2 = _MM_ADD(xLocal_2, xLocal_3);
  xLocal_3 = _MM_MUL(xK2_4, _MM_SET((fptype)-1.821255978));
  xLocal_2 = _MM_ADD(xLocal_2, xLocal_3);
  xLocal_3 = _MM_MUL(xK2_5, _MM_SET((fptype)1.330274429));
  xLocal_2 = _MM_ADD(xLocal_2, xLocal_3);

  xLocal_1 = _MM_ADD(xLocal_2, xLocal_1);
  xLocal  = _MM_MUL(xLocal_1, xNPrimeofX);
  xLocal  = _MM_SUB(_MM_SET((fptype)1.0), xLocal);

  _MM_STORE(OutputX, xLocal);
  // _mm_storel_pd(&OutputX[0], xLocal);
  // _mm_storeh_pd(&OutputX[1], xLocal);

  for (i=0; i<SIMD_WIDTH; i++) {
    if (sign[i]) {
      OutputX[i] = ((fptype)1.0 - OutputX[i]);
    }
  }
}

void BlkSchlsEqEuroNoDiv (fptype * OptionPrice, int numOptions, fptype *
sptprice,
fptype * strike, fptype * rate, fptype * volatility,
          fptype * time, int * otype, float timet)
{
  int i;
// local private working variables for the calculation
  _MMR xStockPrice;
  _MMR xStrikePrice;
  _MMR xRiskFreeRate;
  _MMR xVolatility;
  _MMR xTime;
  _MMR xSqrtTime;

  _MM_ALIGN16 fptype logValues[NCO];
  _MMR xLogTerm;
  _MMR xD1, xD2;
  _MMR xPowerTerm;
  _MMR xDen;
  _MM_ALIGN16 fptype d1[SIMD_WIDTH];
  _MM_ALIGN16 fptype d2[SIMD_WIDTH];
  _MM_ALIGN16 fptype FutureValueX[SIMD_WIDTH];
  _MM_ALIGN16 fptype NofXd1[SIMD_WIDTH];
  _MM_ALIGN16 fptype NofXd2[SIMD_WIDTH];
  _MM_ALIGN16 fptype NegNofXd1[SIMD_WIDTH];
  _MM_ALIGN16 fptype NegNofXd2[SIMD_WIDTH];

  xStockPrice = _MM_LOAD(sptprice);
  xStrikePrice = _MM_LOAD(strike);
  xRiskFreeRate = _MM_LOAD(rate);
  xVolatility = _MM_LOAD(volatility);
  xTime = _MM_LOAD(time);

  xSqrtTime = _MM_SQRT(xTime);

  for(i=0; i<SIMD_WIDTH;i ++) {
    logValues[i] = log(sptprice[i] / strike[i]);
  }

  xLogTerm = _MM_LOAD(&(logValues[0]));

  xPowerTerm = _MM_MUL(xVolatility, xVolatility);
  xPowerTerm = _MM_MUL(xPowerTerm, _MM_SET(0.5));
  // xPowerTerm = _mm_div_pd(xPowerTerm, _MM_SET(2.0, 2.0));

  xD[illegible]te, xPowerTerm);
  xD[illegible]M_SUB(xRiskFreeRate, xPowerTerm);

  xD[illegible]M_MUL(xD1, xTime);
  xD[illegible]);

  xD[illegible]M_ADD(xD1, xLogTerm);
  xD[illegible]M_ADD(xD2, xLogTerm);

  xD[illegible]xSqrtTime);
  xD1 = _MM_DIV(xD1, xDen);
  // VL: 10/15/06. An optimization is not to recompute xD2, but to derive it
from xD1
  // xD2 = _MM_DIV(xD2, xDen);
  xD2 = _MM_SUB(xD1, xDen);

  _MM_STORE(d1, xD1);
  _MM_STORE(d2, xD2);

  CNDF( NofXd1, d1 );
  CNDF( NofXd2, d2 );

  for (i=0; i<SIMD_WIDTH; i++) {
    FutureValueX[i] = strike[i] * (exp(-(rate[i])*(time[i])));

    NegNofXd1[i] = ((fptype)1.0 - (NofXd1[i]));
    NegNofXd2[i] = ((fptype)1.0 - (NofXd2[i]));
    OptionPrice[i] = (FutureValueX[i] * NegNofXd2[i]) - (sptprice[i] *
NegNofXd1[i]);
  }
}

void sseBlackScholes(fptype *option_price,
          int num_options,
          fptype *stkprice,
          fptype *strike,
          fptype *rate,
          fptype *volatility,
          fptype *time)
{
  for (int i = 0; i < num_options; i += NCO) {
    // Calling main function to calculate option value based on Black &
Sholes's
    // equation.
    BlkSchlsEqEuroNoDiv(&(option_price[i]), NCO, &(stkprice[i]),
&(strike[i]),
              &(rate[i]), &(volatility[i]), &(time[i]), NULL/*&(otype[i])*/, 0);
  }
}
```

186 lines

- vectorized
- *not threaded*
- *machine dependent*

intel
Software Products

Intel's Family of Parallel Models

Intel® Parallel Building Blocks (PBB)		Fixed Function Libraries	Established Standards	Research and Exploration
Intel® Threading Building Blocks (TBB)	Intel® Cilk Plus	Intel® Math Kernel Library (MKL)	MPI	Intel® Concurrent Collections
Intel® Array Building Blocks (ArBB)		Intel® Integrated Performance Primitives (IPP)	OpenMP*	OpenCL*

Where do Customers Get them?

Launched Nov 9!

Intel® Parallel Studio XE 2011

Intel® Composer XE 2011

- Intel® C++ Compiler XE 12.0
- Intel® Fortran Compiler XE 12.0
- Intel® Parallel Debugger Extension
- Intel® Parallel Building Blocks (all)
- Intel® Math Kernel Library

Intel® Inspector XE 2011

Intel® Vtune™ Amplifier XE 2011

Windows: Integrates into Microsoft* Visual Studio* or stand-alone
Linux: Integrates into Eclipse CDT
1 Year Premier Support Renewable Annually

Launched Sep 2!

Intel® Parallel Studio 2011

Intel® Parallel Advisor 2011

Intel® Parallel Composer 2011

- Intel® C++ Compiler XE 12.0
- Intel® Parallel Debugger Extension
- Intel® Parallel Building Blocks
- Intel® Threading Building Blocks
- Intel® Cilk™ Plus

Intel® Parallel Inspector 2011

Intel® Parallel Amplifier 2011

Windows: Integrates into Microsoft* Visual Studio*
1 Year Premier Support Renewable Annually

Levels of Parallelism

Grid	Group of clusters communicating through internet
Cluster	Group of computers communicating through fast interconnect
Node	Group of processors communicating through shared memory
Socket	Group of cores communicating through shared cache
Core	Group of functional units communicating through registers
Hyper-Threads	Group of thread contexts sharing functional units
Superscalar	Group of instructions sharing functional units
Pipeline	Sequence of instructions sharing functional units
Vector	Single instruction using multiple functional units

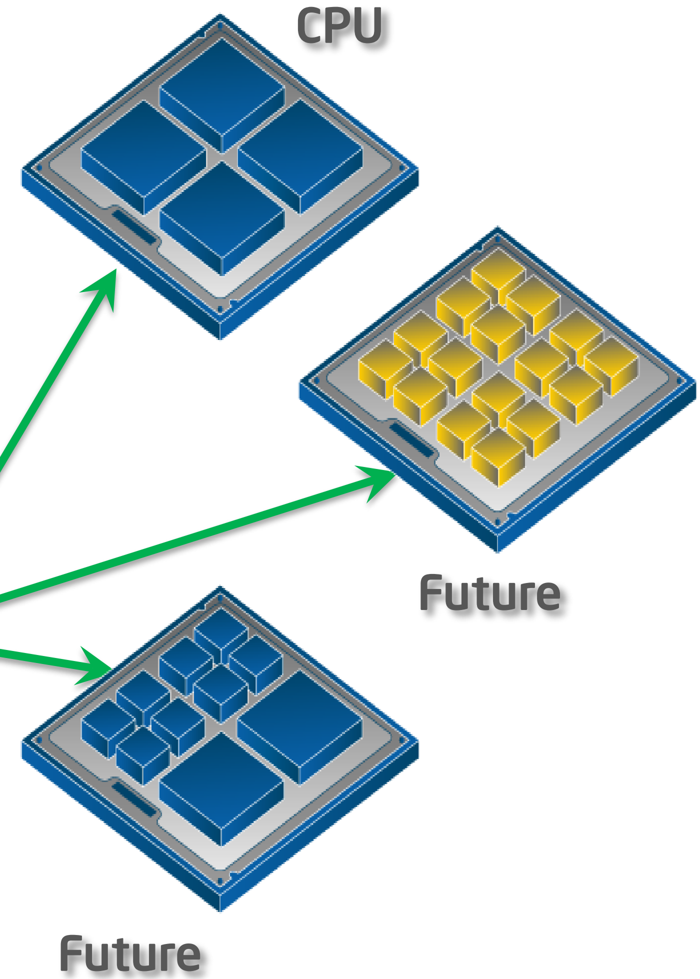
How does it work?

Sequentially consistent semantics

ArBB kernels in "serial" C++ app
- Templates
- Overloaded operators

Standard C++ compiler
- Links with dynamic library

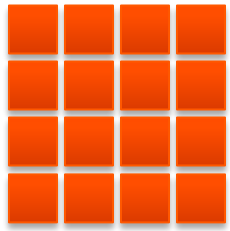
ArBB Runtime
- Dynamic compiler
- Threading and heterogeneous runtime



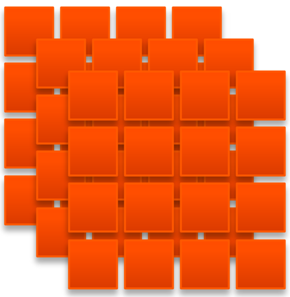
Containers

regular containers

dense<T>

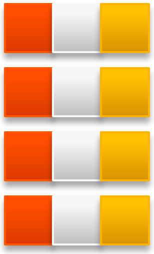


dense<T, 2>



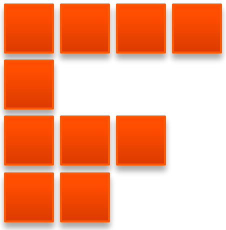
dense<T,3>

array<…>
struct user_type {..};
class user_type { };



dense<array<…>>
dense<user_type>

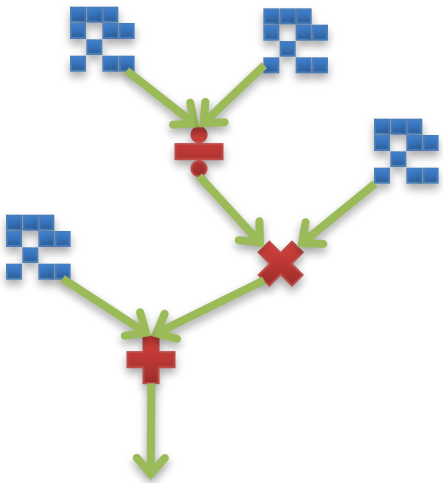
irregular containers



nested<T>

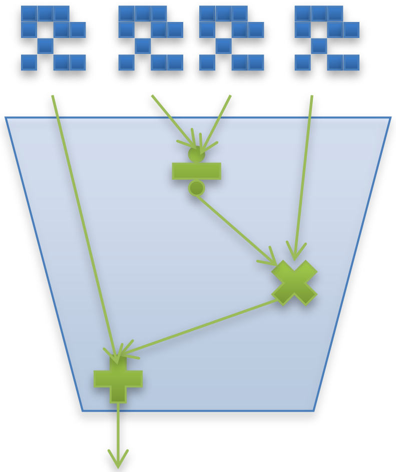
Vector Processing *or* Scalar Processing

Vector Processing



```
dense<f32> A, B, C, D;
A = A + B/C * D;
```

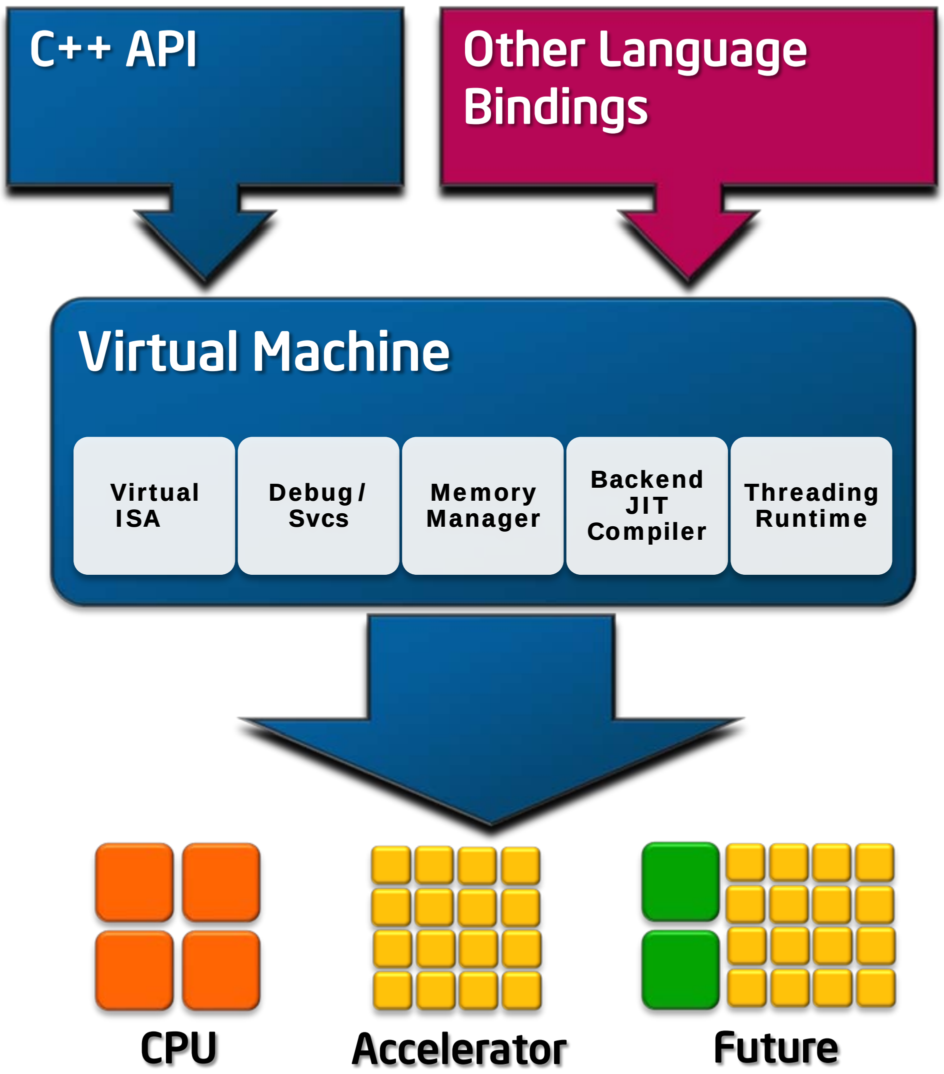
Scalar Processing



```
void kernel(f32& a, f32 b, f32 c, f32 d) {
    a = a + (b/c)*d;
}
...
dense<f32> A, B, C, D;
map(kernel)(A, B, C, D);
```

Intel® ArBB Virtual Machine

- Generalized data-parallel programming model
- Supports wide variety of patterns and collections
- Supports explicit dynamic generation and management of code
- Implementation targets both threads and vector code
  - Machine independent optimization
  - Offload management
  - Machine specific code generation and optimizations
  - Scalable threading runtime



Interface: The API as a Language

Syntax and semantics that extend C++

Adds parallel collection objects and methods to C++

- Uses standard C++ features (classes, simple templates, and operator overloading) to create new types and operators
- Sequences of API calls are fused and optimized by a JIT compiler

Works with standard C++ compilers

- Intel® C++ Compiler
- Microsoft* Visual* C++ Compiler
- GNU Compiler Collection*

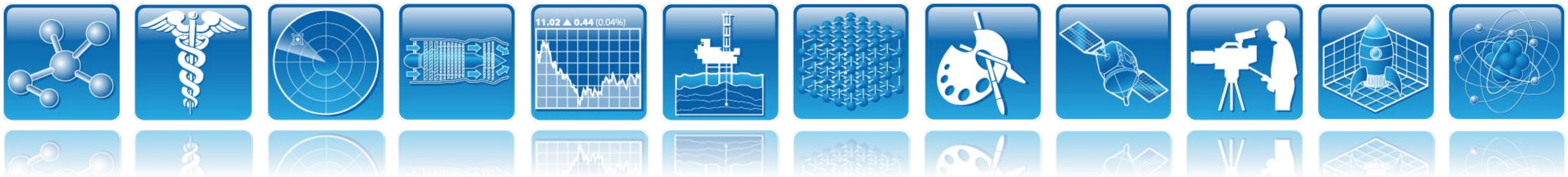
Express algorithms using mathematical notation

- Developers focus on *what to do,* not *how to do it*

Uses sequential semantics

- Developers do not use *threads, locks* or other lower-level constructs and can avoid the associated *complexity*
- *Programmers can reason and debug as if the program were serial.*

What can it be used for?



Bioinformatics

- Genomics and sequence analysis
- Molecular dynamics

Engineering design

- Finite element and finite difference simulation
- Monte Carlo simulation

Financial analytics

- Option and instrument pricing
- Risk analysis

Oil and gas

- Seismic reconstruction
- Reservoir simulation

Medical imaging

- Image and volume reconstruction
- Analysis and computer aided detection (CAD)

Visual computing

- Digital content creation (DCC)
- Physics engines and advanced rendering
- Visualization
- Compression/decompression

Signal and image processing

- Computer vision
- Radar and sonar processing
- Microscopy and satellite image processing

Science and research

- Machine learning and artificial intelligence
- Climate and weather simulation
- Planetary exploration and astrophysics

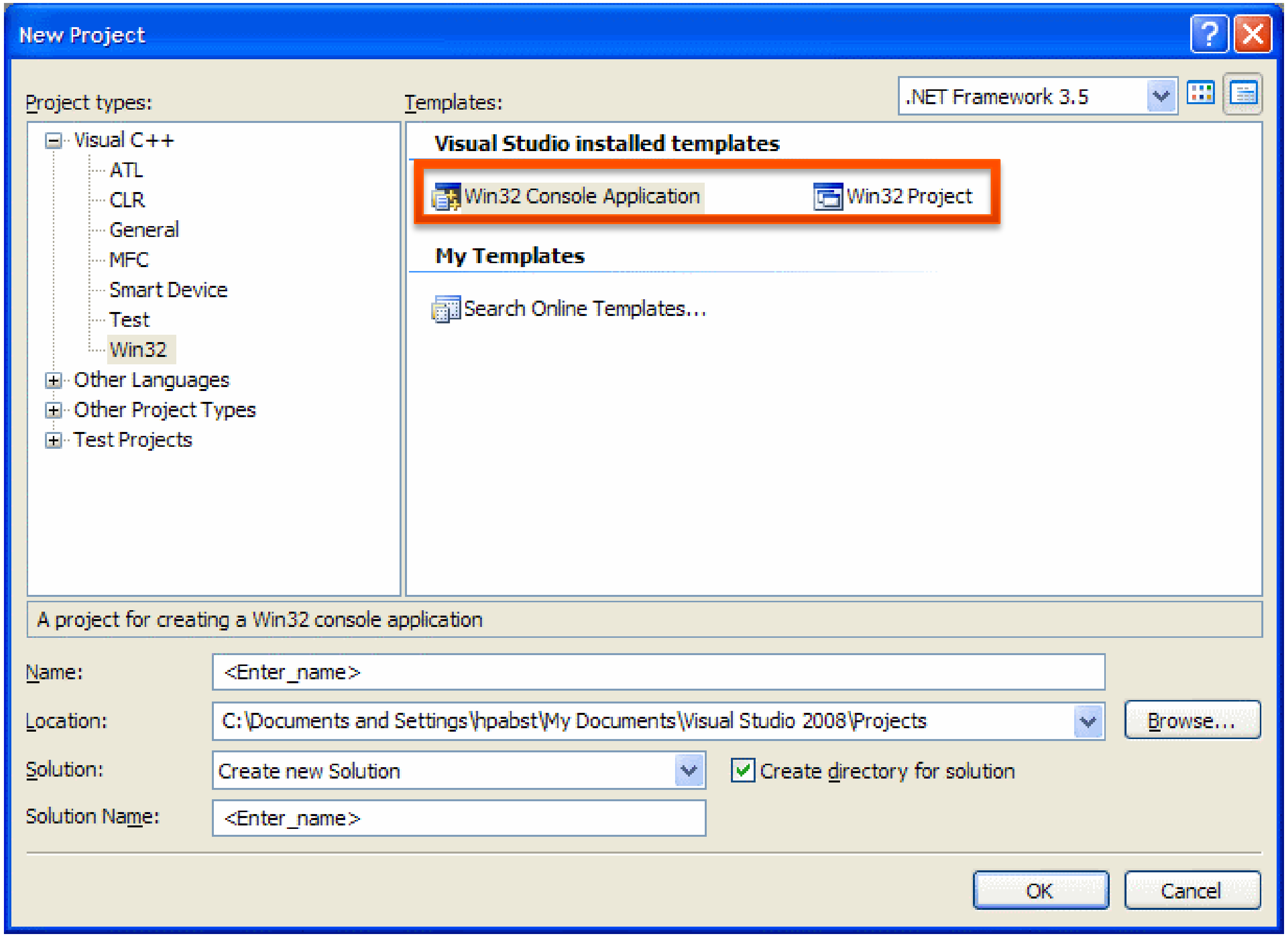
Enterprise

- Database search
- Business information

Introduction to Intel® Array Building Blocks

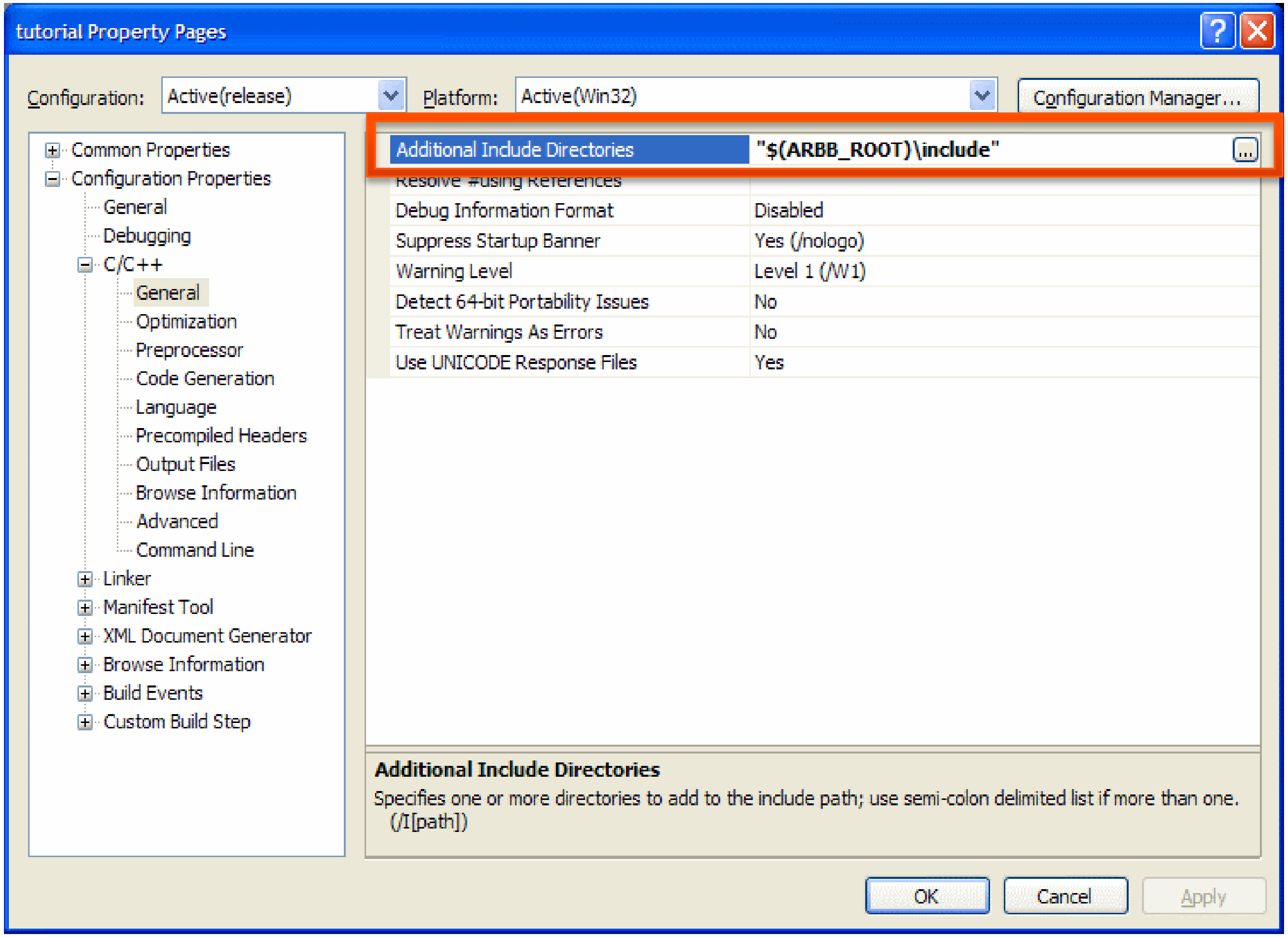
How to add it to your project…

Intel® ArBB in a Visual Studio* Project



Screenshots taken from Microsoft* Visual Studio 2008*

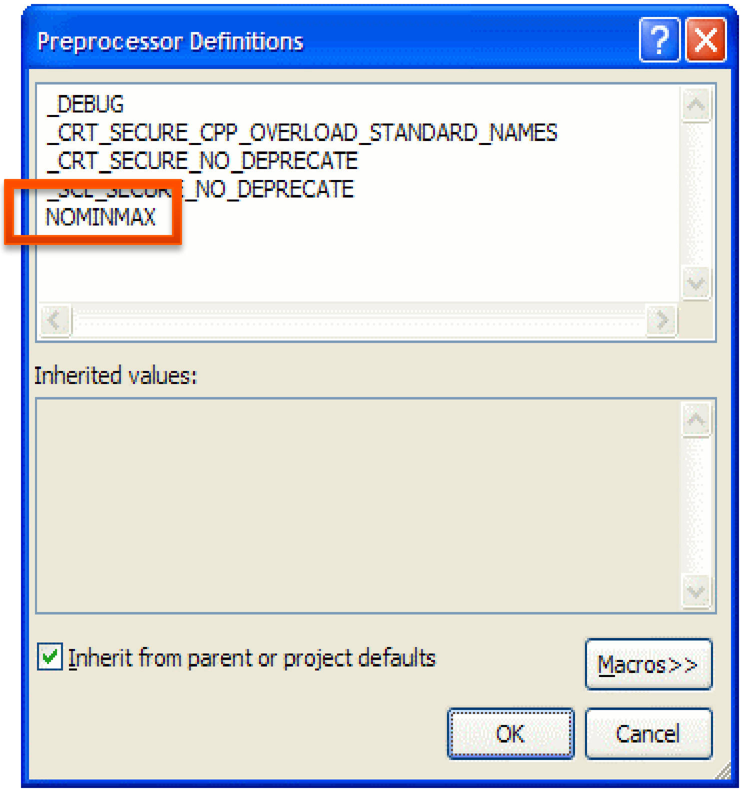
Intel® ArBB in a Visual Studio* Project



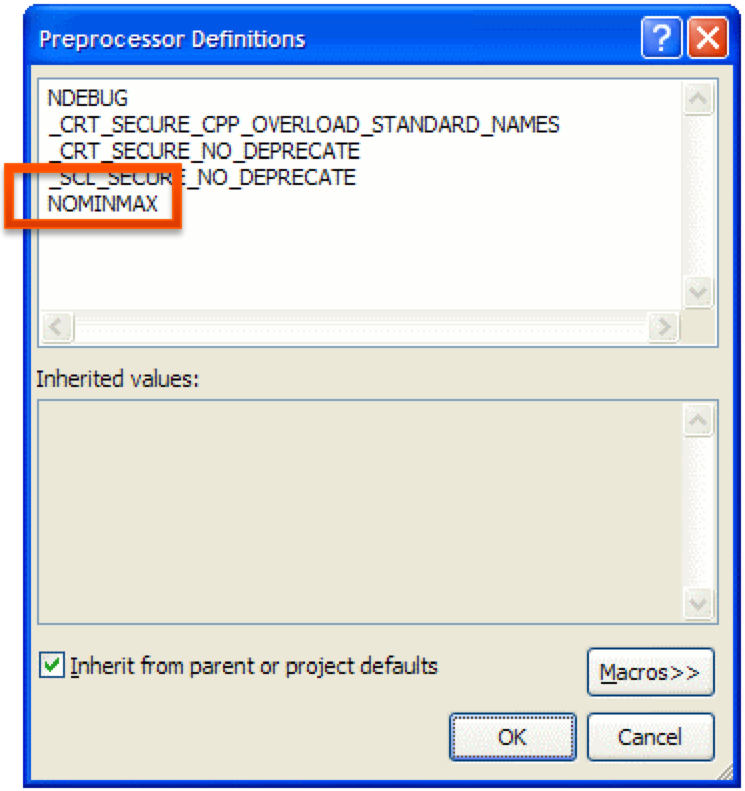
Screenshots taken from Microsoft* Visual Studio 2008*

Including ArBB in a Visual Studio* Project

Debug Mode

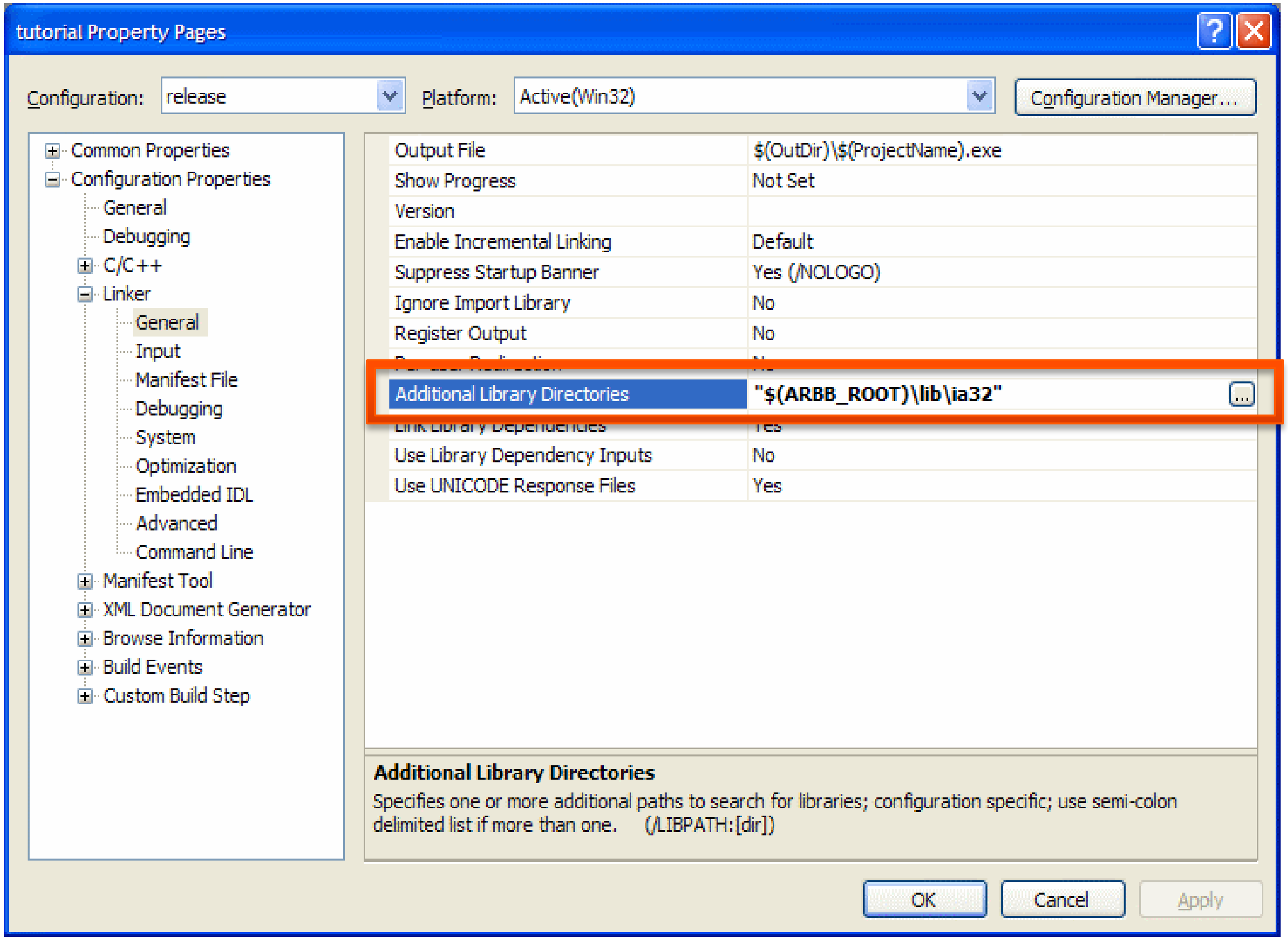


Release Mode



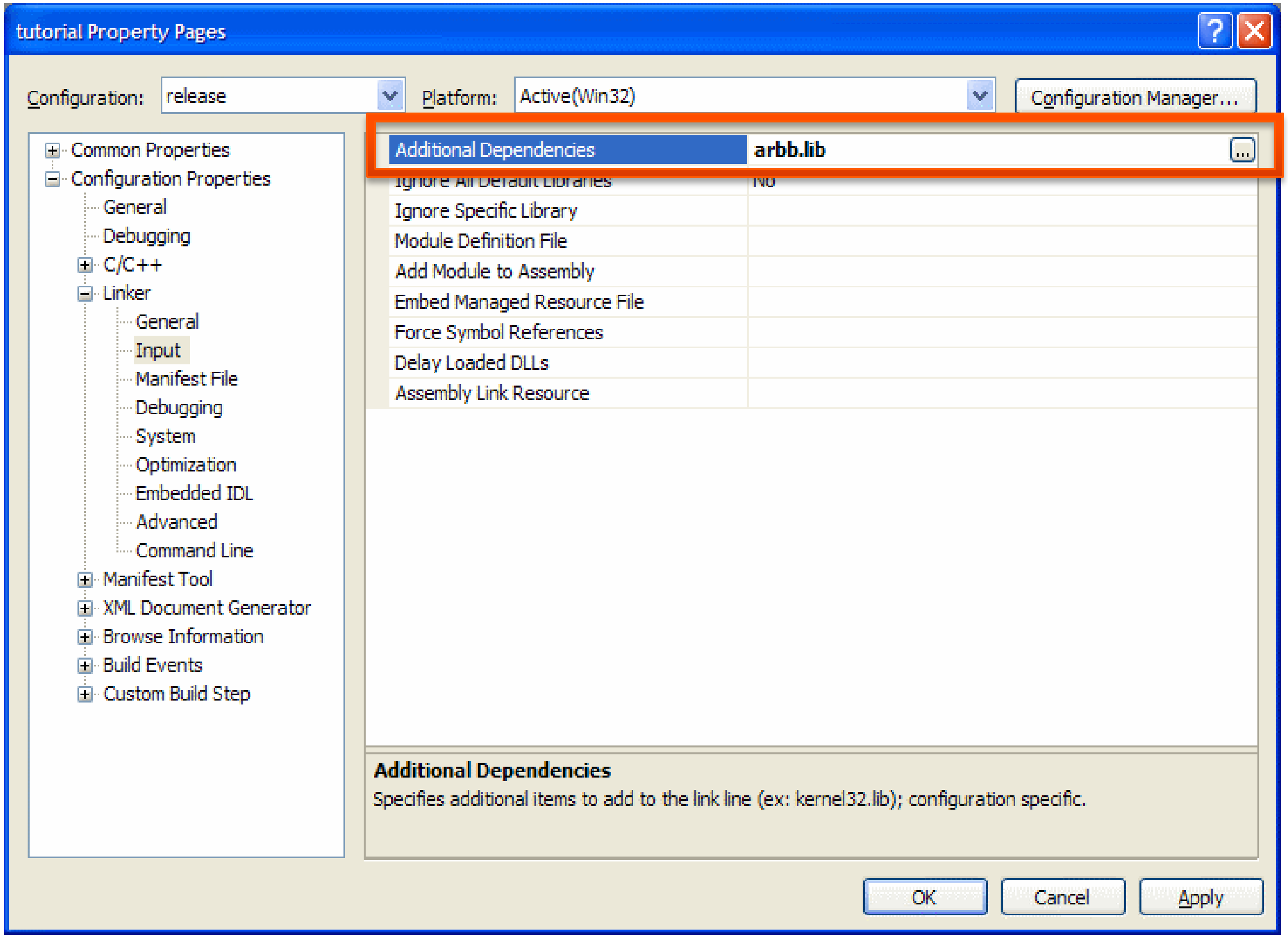
Screenshots taken from Microsoft* Visual Studio 2008*

Intel® ArBB in a Visual Studio* Project



Screenshots taken from Microsoft* Visual Studio 2008*

Intel® ArBB in a Visual Studio* Project



Screenshots taken from Microsoft* Visual Studio 2008*

Intel® ArBB in a Makefile-based Project

- **Make available ArBB include (header) files:**
  `-I/opt/intel/arbb/include`
  (modify compiler search path for include files)

- **Make available ArBB libraries**
  `-L/opt/intel/arbb/lib/{ia32,intel64}`
  (modify linker search path for libraries)

- **Include ArBB libraries in linker process**
  `-larbb –ltbb`

Using the Intel® ArBB API

- Include the definitions

```
#include <arbb.hpp>
```

- Import the namespace or specific identifiers

```
using namespace arbb;
using namespace arbb::add_reduce;
```

- Good practice:
  - To not pollute the name spaces, restrict scope of “using” statement as much as possible, especially in headers
  - Selectively include ArBB names only if used

Code Skeleton for Intel® ArBB Applications

- Use the following code skeleton for ArBB applications

```
int main(int argc, char* argv[]) {
    int ret_code;
    try {
        // call into ArBB code
        ret_code = EXIT_SUCCESS;
    }
    catch(const std::exception& e) {
        ret_code = EXIT_FAILURE;
    }
    catch(...) {
        cerr << "Error: Unknown exception caught!" << endl;
        ret_code = EXIT_FAILURE;
  }
  return ret_code;
}
```

- ArBB indicates runtime errors through standard C++ exceptions
- Existing top-level entry points do not need to change if they already catch std::exception

Introduction to Intel® Array Building Blocks

Programming Constructs and Data Types

Overall Syntax Conventions

- **All Identifiers are lower-case with underscores**
  - some_type
  - some_class::some_member_function()
- **Chosen to align with C++ standard library conventions**

Intel® ArBB Constructs

- **Scalar types**
  - Equivalent to primitive C++ types
- **Vector types**
  - Parallel collections of (scalar) data
- **Operators**
  - Scalar operators
  - Vector operators
- **Functions**
  - User-defined code fragments
- **Control flow constructs**
  - Conditionals, iteration, etc.
  - These are for *serial* control flow *only*
  - Vector operations and "map" are used for expressing parallelism

Scalar types

- **Scalar types provide equivalent functionality to the scalar types built into C/C++**

Types	Description	C++ equivalents
f32, f64	32/64 bit floating point number	float, double
i8, i16, i32, i64	8/16/32 bit signed integers	char, short, int
u8, u16, u32, u64	8/16/32 bit unsigned integers	unsigned char/short/int
boolean	Boolean value (true or false)	bool
usize, isize	Signed/unsigned integers sufficiently large to store addresses	size_t (eqv. usize)

Scalar Types

Use scalar types for ArBB scalar computation

```
i32 int_scalar;                          // a scalar 32-bit integer value
f32 fp_scalar = (f32)int_scalar;         // cast a scalar to new type
```

Casting to/from C/C++ types

```
float f = (float)fp_scalar;              // NOT supported
f32 fp_scalar2(f);                       // immediate copy
f32 fp_scalar3 = f;                      // immediate copy
float x = value(fp_scalar);              // retrieve value
```

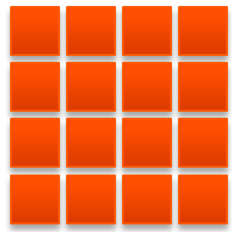
Constant values are supported (types must match)

```
f32 fp_scalar = (f32)int_scalar + 0.5f;
f32 r = 2.0f;
fp_scalar = 3.14f * r * r;
```

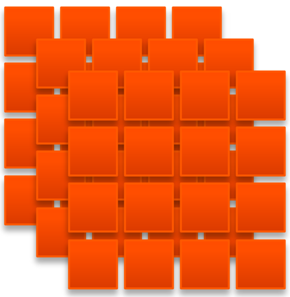
Containers

regular containers

dense<T>

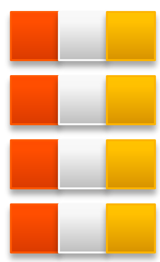


dense<T, 2>



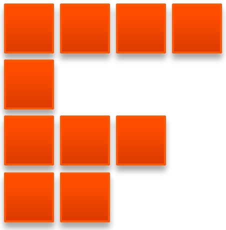
dense<T,3>

array<...>
struct user_type {..};
class user_type { };



dense<array<...>>
dense<user_type>

irregular containers



nested<T>

Dense Containers

```
template<typename T, std::size_t D = 1>
class dense;
```

- **This is the equivalent to `std::vector` or C arrays**
- **Dimensionality is optional, defaults to 1**

Property	Restrictions	Can be set at
Element type	Must be an ArBB scalar or user-defined type	Compile time
Dimensionality	1, 2, or 3	Compile time
Size	Only restricted by free memory	Runtime

Declaration and Construction

Declaration	Element type	Dimensionality	Size
dense<f32> a1;	f32	1	0
dense<f32, 1> a2;	f32	1	0
dense<i32, 2> b;	i32	2	0, 0
dense<f32> c(1000);	f32	1	1000
dense<f32> d(c);	f32	1	1000
dense<i8, 3> e(5, 3, 2);	i8	3	5, 3, 2

Operations on dense Containers

- **All scalar operations can be applied element-wise**
  - Arithmetic and bit operations, transcendentals, etc.
- **Additionally provides container operations:**
  - Indexing, e.g. operator[]
  - Reordering, e.g. shift(), section()
  - Reductions, e.g. sum(), any(), all()
  - Prefix sums, packs, and other data-parallel primitives
  - Property access, e.g. num_rows()
- **Most of these operations run in parallel**
  - For example, if you add two dense containers together, all the individual additions can run in parallel

Moving Data into and out of Containers

- **Dense containers provide two ways to access data:**
  - Iterators
    - read_only_range   iterator to read from the container
    - write_only_range   iterator to write into the container
    - read_write_range   iterator to write/read a container
  - Binding
    - On construction, dense containers can be *bound* (associated) to a particular data location
    - Moves data into and out of that location when required

Creating “dense” Containers

Declaration of a dense container:

```
// create an empty container whose values will be assigned later
dense<f32> temp;
```

vector objects of different base types cast into each other:

```
dense<i32> vi = …;
dense<f32> v = (dense<f32>)vi;
```

Filling “dense” Containers

```
// request write-only access to container
dense<f32> a(1024);
range<f32> range_a = a.write_only_range();
std::fill(range_a.begin(),
          range_a.end(),
          static_cast<f32>(1));

// request read/write access to container
dense<f32> b(1024);
range<f32> range_b = b.read_write_range();
std::fill(range_b.begin(),
          range_b.end(),
          static_cast<f32>(2));
```

Fixed-size Arrays

- Typical usages: pairs of data, RGBA data, CYMK data, etc.
- Use std::array look-a-like
  - std::array is a C++ TR1/C++0x type
  - Will support std::array operations
  - You can manipulate with element-wise, horizontal, swizzling, and other utility operations

```
array<f32, 3> p1, p2, p3;

f32 r = p1[0];                     // std::array operations
p1 = p2 + p3;                      // element-wise operations
f32 sum_p1 = sum(p1);              // horizontal operations
p1 = cat(p2, p3);                  // utility operations
```

Structured Types

- C++ classes and structures can be used relatively normally within ArBB
  - Requires that primitive types be classes in ArBB types (f32, etc.)
  - Supports member functions, class members, overloaded operators, etc.
  - However, virtual functions and pointers are resolved during “capture time” only
  - Overloaded operators are automatically lifted over collections
  - Lifting member functions over collections requires an additional declaration (a macro is provided to help with this)

Structure/Class Example

```
class my_class {
public:
  my_class(f32 location, i32 count);
  my_class operator+(const my_class& other) {
    return my_class(location + other.location,
                    max(count, other.count));
  }
  // other code…
private:
  f32 m_location;
  i32 m_count;
};

dense<my_class> A, B, C;
A = B + C;          // This will use the user-defined operator+!
my_class m = A[5];  // Other interactions work naturally.
```

BREAK

intel®